Phase 3 Report

Unit and Integration Tests:

PathfinderTest

- The Pathfinder class affects the movement of the moving enemies, so it is important to the difficulty/enjoyment of gameplay
- Unit tests:
  - testCanPath()
    - Return true if passing an existing path between demon and enemies, otherwise false.
  - testInRange()
    - Return true if detecting the correct range between demon and enemies, otherwise false.
  - testGetPath()
    - Return a path if finding a path.
- Utilize assertTrue(), assertFalse(), Assert.assertNotNull(), @Before

TimeTest

- The Time class keeps track of how long the game has been running, which is a requirement of the game
- Unit tests:
  - testGetMinutes()
    - Test if the Time class converts seconds to minutes correctly
  - testGetSeconds()
    - Test if the Time class converts seconds to minutes and seconds correctly
- Utilize assertEquals(), @Before

ScoreTest

- The Score class keeps track the player's score, which is a requirement of the game and also increases the game's entertainment value (comparing scores from different playthroughs)
- Unit tests:
  - increaseScore()
    - Adding multiple values together correctly
  - decreaseScore()
    - Subtracting multiple values correctly
  - addSubtract()
    - Correctly adding and subtracting numbers in tandem
- Integration tests:
  - interactionGameWin()
    - Interactions with GameManager class's win variable (when score < 0 and when score > 0)

        - interactionGameDone()
            - Interactions with GameManager class's gameDone variable (when score < 0 and when score > 0)
    - Utilize assertTrue(), assertFalse(), assertEquals()

TrapTest

- The Trap class provides secondary obstacles for the player to increase game enjoyment, and also is another way for the player to lose the game
- Unit tests:
    - checkRewardPoints()
        - Check if penalty set as expected (-110 points)
- Integration tests:
    - checkScoreInteraction()
        - Check interaction with Score and GameManager classes
            - If too many traps are collected, do the game-end (gameDone) and the player-loss (win) conditions/variables get triggered/updated?
- Utilize assertTrue(), assertFalse(), assertEquals(), @Before

RegularRewardTest

- The regular rewards hold the key to winning the game and help prevent the player from losing (providing a safety net for running into traps), so it is important that they work correctly
- Unit tests:
    - checkRegularRewardPoints()
        - Check if regular reward points set as expected (100 points)
- Integration tests:
    - checkScoreInteraction()
        - Check interaction with Score class
            - If the player collects the reward, does their score increase?
- Utilize assertEquals(), @Before

BonusRewardTest

- The bonus reward increases the game's replayability (appears in a different spot on the map every time) and enjoyment (comparing scores), so it is important that it works correctly
- Unit tests:
    - checkRewardPoints()
        - Check if bonus reward points set as expected (200 points)
- Integration tests:
    - checkScoreInteraction()
        - Check interaction with Score class
            - If the player collects the reward, does their score increase?
- Utilize assertEquals(), @Before

Test Quality and Coverage:

Measures taken to ensure the quality of the test cases:

We each wrote our own tests, then had the other group members run and look them over to remove any personal bias and make sure that our logic was sound and our tests solid and strict.

Coverage of tests:

- Line:
    - ~35.09%
        - Total lines of code: 818
            - 9 (wall) + 12 (trap) + 22 (time) + 18 (score) + 12 (regular reward) + 92 (pathfinder) + 64 (maze) + 5 (main) + 29 (game obj) + 129 (game manager) + 29 (enemies) + 93 (menu frame) + 79 (end frame) + 7 (door) + 75 (demon) + 117 (credit frame) + 26 (bonus reward)
        - Lines covered: 287
            - Amount of different lines covered (did not for overlap (necessary but similar code is executed each time an instance of GameManager is created))
- Branch:
    - ~40.06%

We performed both line and branch tests as required, but also performed a series of system tests to get a better idea of how our game is working. Some of those system tests were performance and stress testing, since all our group members have different devices and use different software. In these tests, one of the things we did was use the game loop (the game thread) in the GameManager class to check the average amount of time our game needed to update all the game objects (player character, moving enemies, bonus reward, etc.), where we updated the FPS of our game accordingly so as to standardize the amount of time between each update and to get the best performance without stressing out the system.

Features or code segments that are not covered:

Some classes (like Door and especially the Menu, End, and Credit frames) only contained their constructors, so they were not explicitly tested, especially since creating an instance of the GameManager class would automatically create an instance of the object due to the way GameManager is initialized (Line 65 in the GameManager creates a Door object); no exceptions are ever thrown when the GameManager is initialized, so we decided to focus on classes that needed more attention, like the Score and Pathfinder classes. It is just unfortunate that the Menu, End, and Credit frames contain an average of 100 lines of code each, which skews our line coverage percentage.

We also did not cover methods with a void return type, as they cannot be asserted or are very difficult to test (cannot test local variables in a function). The bulk of the lines and branches of our code were in these types of methods, such as update() in the BonusReward class, and the draw() methods of classes like Time and Score, hence our coverage percentages.

Findings:

During the process of writing and running tests using JUnit with Maven in Java, our team made several important findings. We were able to reveal and fix bugs in the production code, which ultimately improved the overall quality of our code.

One of the key findings was that thorough testing using JUnit helped us identify and fix several critical bugs that would have otherwise gone unnoticed. The tests acted as a safety net, allowing us to catch and rectify issues before they made their way into production. This helped us deliver a more stable and reliable software product to our end-users.

Additionally, during the testing phase, we made necessary changes to the production code to ensure it met the desired functionality and performance requirements. For example, we created new getter methods in certain classes to make them more easily testable (BonusReward, Trap, etc.). This allowed us to write more comprehensive tests and ensure that the code was behaving as expected.

The JUnit tests also provided us with valuable feedback on the design and architecture of our code. We identified areas where the code was tightly coupled, leading to issues with maintainability and scalability. By refactoring these areas and improving the test coverage, we were able to enhance the overall quality of our codebase.

In conclusion, the use of JUnit with Maven in Java for testing proved to be an effective approach for identifying and fixing bugs, improving code quality, and validating the functionality and performance of our software. The findings from our testing efforts led to necessary changes in the production code, such as creating new getter methods to make certain classes more easily testable. This resulted in a more robust and reliable software product. Overall, the testing phase was a crucial step in our software development process, helping us deliver a higher-quality and more reliable software solution to our end-users. Additionally, it allowed us to uncover and address issues related to code design and architecture, and identify and fix edge cases that were not initially considered. This demonstrates the value of thorough testing in improving the overall quality of our codebase.